TypeScript相关笔记

目 录

1. 简介
   1. 什么是TypeScript

TypeScript 是 JavaScript 的一个超集，主要提供了类型系统和对 ES6 的支持，它由 Microsoft 开发。

TypeScript 是 JavaScript 的类型的超集，它可以编译成纯 JavaScript。编译出来的 JavaScript 可以运行在任何浏览器上。TypeScript 编译工具可以运行在任何服务器和任何系统上。TypeScript 是开源的。

* + 1. 为什么选择 TypeScript
       1. TypeScript 增加了代码的可读性和可维护性

类型系统实际上是最好的文档，大部分的函数看看类型的定义就可以知道如何使用了

可以在编译阶段就发现大部分错误，这总比在运行时候出错好

增强了编辑器和 IDE 的功能，包括代码补全、接口提示、跳转到定义、重构等

* + - 1. TypeScript 非常包容

TypeScript 是 JavaScript 的超集，.js 文件可以直接重命名为 .ts 即可

即使不显式的定义类型，也能够自动做出类型推论

可以定义从简单到复杂的几乎一切类型

即使 TypeScript 编译报错，也可以生成 JavaScript 文件

兼容第三方库，即使第三方库不是用 TypeScript 写的，也可以编写单独的类型文件供 TypeScript 读取

* + - 1. TypeScript 拥有活跃的社区

大部分第三方库都有提供给 TypeScript 的类型定义文件

Google 开发的 Angular2 就是使用 TypeScript 编写的

TypeScript 拥抱了 ES6 规范，也支持部分 ESNext 草案的规范

* + 1. TypeScript 的缺点

任何事物都是有两面性的，我认为 TypeScript 的弊端在于：

有一定的学习成本，需要理解接口（Interfaces）、泛型（Generics）、类（Classes）、枚举类型（Enums）等前端工程师可能不是很熟悉的概念

短期可能会增加一些开发成本，毕竟要多写一些类型的定义，不过对于一个需要长期维护的项目，TypeScript 能够减少其维护成本

集成到构建流程需要一些工作量

可能和一些库结合的不是很完美

* 1. 安装TypeScript
     1. 安装

TypeScript 的命令行工具安装方法如下：

npm install -g typescript

以上命令会在全局环境下安装。

通过如下命令判断是否安装成功(显示版本号)：

tsc -v

安装完成之后，我们就可以在任何地方执行 tsc 命令了。

* + 1. 编辑器

Visual Studio Code

插件：TypeScript Extension Pack

https://blog.csdn.net/qq\_34414916/article/details/85156499

vscode下搭建Typescript编译环境

https://www.jianshu.com/p/4f219f20af75

点击【任务】-【运行任务】

之后会出现tsc:build 、tsc:watch两个选项

-tsc:build选项是用于一次编译。

-tsc:watch选项可以监测ts文件的改动，可以进行实时编译。

经过编译后的文件会存放在之前配置好的js文件目录下。

* + 1. 学习

官方文档：

https://www.tslang.cn/docs/handbook/typescript-in-5-minutes.html

https://ts.xcatliu.com/introduction/hello-typescript

http://www.ruanyifeng.com/blog/2015/02/strong-typing-javascript.html

https://www.jianshu.com/p/70a03e21936c

https://www.tslang.cn/docs/handbook/basic-types.html

https://www.w3cschool.cn/typescript/

* 1. 类型

typescript中为了使编写的代码更规范，更有利于维护，增加了类型校验

写ts代码必须指定类型：13种类型

布尔类型（Boolean）

数字类型（Number）

字符串类型(String)

标记类型（Symbol）（暂时不支持）

数组类型（Array）

元组类型（Tuple）

枚举类型（Enum）

任意类型（Any）

空值类型（Void）

null 和 undefined

非存在类型（Never）

非原始类型（Object）

类型断言

* + 1. 布尔类型（Boolean）

最基本的数据类型就是简单的true/false值，在JavaScript和TypeScript里叫做boolean（其它语言中也一样）。

let flag: boolean = true

flag = 12345 // 错误

flag = false // 正确

* + 1. 数字类型（Number）

和JavaScript一样，TypeScript里的所有数字都是浮点数。这些浮点数的类型是number。除了支持十进制和十六进制字面量，TypeScript还支持ECMAScript2015中引入的二进制和八进制字面量。

let decimal: number = 6

let hex: number = 0xf00d

let binary: number = 0b1010

let octal: number = 0o744

/\* js \*/

var decimal = 6

var hex = 0xf00d

var binary = 10

var octal = 484

* + 1. 字符串类型(String)

JavaScript程序的另一项基本操作是处理网页或服务器端的文本数据。像其它语言里一样，我们使用string表示文本数据类型。和JavaScript一样，可以使用双引号（"）或单引号（'）表示字符串。

你还可以使用模版字符串，它可以定义多行文本和内嵌表达式。这种字符串是被反引号包围（ `），并且以${expr}这种形式嵌入表达式。

let color: string = "blue"

color = 'red'

let fullName: string = `Bob Bobbington`

let age: number = 37

let sentence1: string = `Hello, my name is ${ fullName }. I'll be ${ age + 1 } years old next month.`

// 和上文效果类似

let sentence2: string = "Hello, my name is " + name + ".\n\n" + "I'll be " + (age + 1 ) + " years old next month."

* + 1. 标记类型（Symbol）（暂时不支持）

自ECMAScript2015起，symbol成为了一种新的原生类型，就像number和string一样。symbol类型的值是通过Symbol构造函数创建的。

let sym1 = Symbol()

let sym2 = Symbol("key")

// 可选的字符串key

Symbols是不可改变且唯一的。

let sym3 = Symbol("key")

let sym4 = Symbol("key")

sym3 === sym4 // false, symbols是唯一的

// 像字符串一样， symbols也可以被用做对象属性的键。

let sym = Symbol()

let obj = {[sym]: "value"}

console.log(obj[sym]); // "value"

Symbols也可以与计算出的属性名声明相结合，来声明对象的属性和类成员。

const getClassNameSymbol = Symbol()

class C {

[getClassNameSymbol](){

return "C"

}

}

let c = new C()

let className = c[getClassNameSymbol]() // "C"

* + 1. 数组类型（Array）

TypeScript像JavaScript一样可以操作数组元素。有两种方式可以定义数组。

第一种，可以在元素类型后面接上[]，表示由此类型元素组成的一个数组：

let list1: number[] = [1, 2, 3]

第二种方式是使用数组泛型，Array<元素类型>：

let list2: Array<number>= [1, 2, 3]

* + 1. 元组类型（Tuple）

元组类型允许表示一个已知元素数量和类型的数组，各元素的类型不必相同。比如，你可以定义一对值分别为string和number类型的元组。

// Declare a tuple type

let x: [string, number];

// Initialize it

x = ["hello", 10]; // OK

// Initialize it incorrectly

x = [10, "hello"]; // Error

当访问一个已知索引的元素，会得到正确的类型：

console.log(x[0].substring(1)); // OK

console.log(x[1].substring(1)); // Error, 'number' does not have 'substring'

当访问一个越界的元素，会使用联合类型替代：

x[3] = "world"; // Error, Property '3' does not exist on type '[string, number]'.

console.log(x[5].toString()); // Error, Property '5' does not exist on type '[string, number]'.

* + 1. 枚举类型（Enum）

enum类型是对JavaScript标准数据类型的一个补充。像C#等其它语言一样，使用枚举类型可以为一组数值赋予友好的名字。

enum Color{ Red, Green, Blue }

let c: Color= Color.Green // 1

默认情况下，从0开始为元素编号。你也可以手动的指定成员的数值。例如，我们将上面的例子改成从1开始编号：

enum Color{ Red= 1, Green, Blue }

let c: Color = Color.Green // 2

或者，全部都采用手动赋值：

enum Color{ Red= 1, Green= 2, Blue= 4 }

let c: Color = Color.Green // 2

枚举类型提供的一个便利是你可以由枚举的值得到它的名字。例如，我们知道数值为2，但是不确定它映射到Color里的哪个名字，我们可以查找相应的名字：

enum Color{ Red = 1, Green, Blue }

let colorName: string= Color[2]

console.log(colorName) // 显示'Green'因为上面代码里它的值是2

指定枚举值为字符串

const { log } = console

enum Color{ Red = 'rRed', Green = 'rGreen', Blue = 'rBlue' }

const c: Color = Color.Red // 1

log(c) // rRed

log(Color.Green) // rGreen

log(Color.Blue) // rBlue

请注意这边如果将枚举值指定为非number类型的值，则需要每个枚举值都进行手动赋值，否则就会报错Enummember must have initializer.。

// Enum member must have initializer.

enum Color{ Red = 'rRed', Green, Blue }

请注意这边如果将枚举值指定为非number类型的值，则需要每个枚举值都进行手动赋值，同时，除了number类型，手动赋值的类型需要为同一种类型。

// fine

enum Color{ Red = 0, Green = 'rGreen', Blue = 'rBlue' }

// Computed values are not permitted in an enum with string valued members.

enum Color{ Red = true, Green = 'rGreen', Blue = 'rBlue' }

指定枚举值为布尔值

// Type 'true' is not assignable to type 'Color'

// Type 'false' is not assignable to type 'Color'

enum Color{ Red = 0, Green = true, Blue = false }

指定枚举值为null和undefined

// Type 'null' is not assignable to type 'Color'

// Type 'undefined' is not assignable to type 'Color'

enum Color{ Red = 0, Green = null, Blue = undefined }

指定枚举值为array和object

// Type 'never[]' is not assignable to type 'Color'

// Type '{}' is not assignable to type 'Color'

enum Color{ Red = 0, Green = [], Blue = {} }

指定枚举值为symbol

const sys = Symbol()

// Type 'unique symbol' is not assignable to type 'Color'.

// Type '{}' is not assignable to type 'Color'

enum Color{ Red = 0, Green = sys, Blue }

* + 1. 任意类型（Any）

有时候，我们会想要为那些在编程阶段还不清楚类型的变量指定一个类型。这些值可能来自于动态的内容，比如来自用户输入或第三方代码库。这种情况下，我们不希望类型检查器对这些值进行检查而是直接让它们通过编译阶段的检查。那么我们可以使用any类型来标记这些变量

let notSure: any = 4

notSure = "maybe a string instead"

notSure = false

// okay, definitely a boolean

在对现有代码进行改写的时候，any类型是十分有用的，它允许你在编译时可选择地包含或移除类型检查。

你可能认为Object有相似的作用，就像它在其它语言中那样。

但是Object类型的变量只是允许你给它赋任意值，但是却不能够在它上面调用任意的方法，即便它真的有这些方法：

let notSure: any = 4

notSure.ifItExists() // TSLint will not throw Error, okay, ifItExists might exist at runtime

notSure.toFixed() // TSLint will not throw Error, okay, toFixed exists (but the compiler doesn't check)

let prettySure: Object = 4

prettySure.toFixed() // TSLint will throw Error: Property 'toFixed' does not exist on type 'Object'.

当你只知道一部分数据的类型时，any类型也是有用的。比如，你有一个数组，它包含了不同的类型的数据

let list: any[] = [1, true, "free"]

list[1] = 100

* + 1. 空值类型（Void）

某种程度上来说，void类型像是与any类型相反，它表示没有任何类型。当一个函数没有返回值时，你通常会见到其返回值类型是void：

function warnUser(): void{

console.log("This is my warning message")

}

声明一个void类型的变量没有什么大用，因为你只能为它赋予undefined和null：

let unusable: void = undefined

fine写法

// fine

function testBreak(): void{

while (true) { break }

}

// fine

function testIf(): void{

if (true) { }

}

// fine

function testDoWhile(x: number): void{

do { } while (x > 3)

}

// fine

function testAny(test: any): void{

return test // fine

}

const { log } = console

log(testAny(1)) // 1

error写法

function testString(test: string): void{

return test // Type 'string' is not assignable to type 'void'.

}

function testNumber(test: number): void{

return test // Type 'number' is not assignable to type 'void'.

}

function testBoolean(test: boolean): void{

return test // Type 'boolean' is not assignable to type 'void'.

}

function testUndefined(test: undefined): void{

return test // Type 'undefined' is not assignable to type 'void'.

}

function testNull(test: null): void{

return test // Type 'null' is not assignable to type 'void'.

}

* + 1. null 和 undefined

TypeScript里，undefined和null两者各自有自己的类型分别叫做undefined和null。和void相似，它们的本身的类型用处不是很大。

// Not much else we can assign to these variables!

let u: undefined = undefined;

let n: null = null;

默认情况下null和undefined是所有类型的子类型。就是说你可以把null和undefined赋值给number类型的变量。然而，当你指定了--strictNullChecks标记，null和undefined只能赋值给void和它们各自。

// null

const x1: string = null // Type 'null' is not assignable to type 'string'.

const x2: number = null // Type 'null' is not assignable to type 'number'.

const x3: boolean = null // Type 'null' is not assignable to type 'boolean'.

const x4: undefined = null // Type 'null' is not assignable to type 'undefined'.

const x5: never = null // Type 'null' is not assignable to type 'never'.

const x6: any = null

const x7: null = null

// undefined

const x1: string = undefined // Type 'undefined' is not assignable to type 'string'.

const x2: number = undefined // Type 'undefined' is not assignable to type 'number'.

const x3: boolean = undefined // Type 'undefined' is not assignable to type 'boolean'.

const x4: null = undefined // Type 'undefined' is not assignable to type 'null'.

const x5: never = undefined // Type 'undefined' is not assignable to type 'never'.

const x6: any = undefined

const x7: undefined = undefined

这能避免很多常见的问题。也许在某处你想传入一个string或null或undefined，你可以使用联合类型string | null | undefined。

注意：我们鼓励尽可能地使用--strictNullChecks，但在本手册里我们假设这个标记是关闭的。

* + 1. 非存在类型（Never）

never类型表示的是那些永不存在的值的类型。例如，never类型是那些总是会抛出异常或根本就不会有返回值的函数表达式或箭头函数表达式的返回值类型；变量也可能是never类型，当它们被永不为真的类型保护所约束时。下面是一些返回never类型的函数：

function testNever(test: never): never{

return test

}

// 返回never的函数必须存在无法达到的终点

function error(message: string): never {

throw new Error(message)

}

// 推断的返回值类型为never

function fail() {

return error("Something failed")

}

// 返回never的函数必须存在无法达到的终点

function infiniteLoop(): never {

while (true) { }

}

function testDoWhile(): never {

do { } while (true)

}

never类型是任何类型的子类型，也可以赋值给任何类型；然而，没有类型是never的子类型或可以赋值给never类型（除了never本身之外）。即使any也不可以赋值给never。

const x1: string = never // 'never' only refers to a type, but is being used as a value here.

const x2: number = never // 'never' only refers to a type, but is being used as a value here.

const x3: boolean = never // 'never' only refers to a type, but is being used as a value here.

const x4: null = never // 'never' only refers to a type, but is being used as a value here.

const x5: never = never // 'never' only refers to a type, but is being used as a value here.

const x6: any = never // 'never' only refers to a type, but is being used as a value here.

const x7: undefine = never // 'never' only refers to a type, but is being used as a value here.

返回never的函数必须存在无法达到的终点

// A function returning 'never' cannot have a reachable end point.

function testBreak(): never {

while (true) { break }

}

// A function returning 'never' cannot have a reachable end point.

function testIf(): never {

if (true) { }

}

// A function returning 'never' cannot have a reachable end point.

function testDoWhile(x:number): never {

do { } while (x > 3)

}

function testString(test: string): never{

return test // Type 'string' is not assignable to type 'never'.

}

function testNumber(test: number): never{

return test // Type 'number' is not assignable to type 'never'.

}

function testBoolean(test: boolean): never{

return test // Type 'boolean' is not assignable to type 'never'.

}

function testUndefined(test: undefined): never{

return test // Type 'undefined' is not assignable to type 'never'.

}

function testNull(test:null):never{

return test // Type 'null' is not assignable to type 'never'.

}

function testAny(test:any):never{

return test // Type 'any' is not assignable to type 'never'.

}

* + 1. 非原始类型（Object）

object表示非原始类型，也就是除number，string，boolean，symbol，null或undefined之外的类型。使用object类型，就可以更好的表示像Object.create这样的API。

例如：

declare function create(o: object | null): void

create({ prop: 0 }) // OK

create(null) // OK

create(42) // Error

create("string") // Error

create(false) // Error

create(undefined) // Error

* + 1. 类型断言

有时候你会遇到这样的情况，你会比TypeScript更了解某个值的详细信息。通常这会发生在你清楚地知道一个实体具有比它现有类型更确切的类型。通过类型断言这种方式可以告诉编译器，“相信我，我知道自己在干什么”。类型断言好比其它语言里的类型转换，或者类型装箱，但是不进行特殊的数据检查和解构。它没有运行时的影响，只是在编译阶段起作用。TypeScript会假设你已经进行了必须的检查。类型断言有两种形式。

其一是“尖括号”语法：

let someValue: any = "this is a string"

let strLength: number = (<string>someValue).length

// 当然直接写也是可以的。

let strLength2: number = someValue.length // 16

另一个为as语法：

let someValue: any = "this is a string"

let strLength: number = (someValue as string).length

两种形式是等价的。至于使用哪个大多数情况下是凭个人喜好；

然而，当你在TypeScript里使用JSX时，只有as语法断言是被允许的。

* 1. 接口
     1. 接口的作用

在面向对象的编程中，接口是一种规范的定义，它定义了行为和动作的规范，在程序设计里面，接口起到一种限制和规范的作用。接口定义了某一批类所需要遵守的规范，接口不关心这些类的内部状态数据，也不关心这些类里方法的实现细节，它只规定这批类里必须提供某些方法，提供这些方法的类就可以满足实际需要。 typescrip中的接口类似于java，同时还增加了更灵活的接口类型，包括属性、函数、可索引和类等。

* + 1. typeScript中的接口

属性类接口

函数类型接口

可索引接口

类类型接口

接口扩展

* + - 1. 可选属性

接口里的属性不全都是必需的。 有些是只在某些条件下存在，或者根本不存在。 可选属性在应用“option bags”模式时很常用，即给函数传入的参数对象中只有部分属性赋值了。

带有可选属性的接口与普通的接口定义差不多，只是在可选属性名字定义的后面加一个?符号。

可选属性的好处之一是可以对可能存在的属性进行预定义，好处之二是可以捕获引用了不存在的属性时的错误。

* + - 1. 只读属性

一些对象属性只能在对象刚刚创建的时候修改其值。 你可以在属性名前用 readonly来指定只读属性:

TypeScript具有ReadonlyArray<T>类型，它与Array<T>相似，只是把所有可变方法去掉了，因此可以确保数组创建后再也不能被修改

* + - 1. readonly vs const

最简单判断该用readonly还是const的方法是看要把它做为变量使用还是做为一个属性。 做为变量使用的话用 const，若做为属性则使用readonly。